OSProj5 Designing a Thread Pool & Producer-Consumer Problem

by 潘禧辰，518021910497

Menu

OSProj5 Designing a Thread Pool & Producer-Consumer Problem
- Menu
- Abstract
- Environment
- Quick Start
  - 编译
    - Designing a Thread Pool
    - Producer-Consumer Problem
  - 测试代码
    - Designing a Thread Pool
    - Producer-Consumer Problem
- Implementation & Result
  - Designing a Thread Pool
    - 数据结构
    - pool_init
    - worker
    - execute
    - pool_submit
    - enqueue
    - dequeue
    - pool_shutdown
    - 完整代码
    - 结果
  - Producer-Consumer Problem
    - buffer implementation
    - main function
    - 结果
- Difficulties
- Reference

Abstract

- 编写了一个thread pool，使用size为3的线程池完成了20个task
- 编写运行了一种Producer-Consumer Problem情况

Environment

- Ubuntu 18.04
- Linux 5.3.0-42-generic
- VMware Workstation Rro 15.5.0 build-14665864

Quick Start

编译

Designing a Thread Pool

Designing a Thread Pool部分osc10e已经给出了Makefile源码，直接使用`make`命令进行编译。

```
make
```

Producer-Consumer Problem

Producer-Consumer Problem使用gcc进行编译即可。

```
gcc buffer.c main.c -lpthread -o main
```

测试代码

Designing a Thread Pool

运行生成的`example`即可

```
./example
```

其中测试数据定义在`client.c`中，进行初始化后进行20个work的测试，再shutdown掉pool，代码如下：

```
#define QUEUE_SIZE 10
int main(void)
{
    // create some work to do
    struct data work[2 * QUEUE_SIZE];
    pool_init();
    for (int i = 0; i < 2 * QUEUE_SIZE; i++)
    {
        work[i].a = i;
        work[i].b = 2 * i;
        pool_submit(&add, &work[i]);
    }
    sleep(1);
    pool_shutdown();
    return 0;
}
```

Producer-Consumer Problem

测试使用了3秒的总时长，2个生产者进程和2个消费者进程。

```
./main 3 2 2
```

Implementation & Result

Designing a Thread Pool

`client.c`中定义了测试数据，可以refer前边测试代码部分。

数据结构

定义了如下数据结构。

- `cu_task`用于在`worker`函数中接收`dequeue`的返回值
- `thread_pool`用于存储线程池的thread id
- `work_queue`用于存储任务队列
- `mutex`为mutex
- `semaphore`为semaphore
- `length`用于表示任务队列长度
- `on`用于标识当前线程池状态，0时代表未运行，1代表正在运行

```
task cu_task;
pthread_t thread_pool[NUMBER_OF_THREADS];
task work_queue[QUEUE_SIZE];
pthread_mutex_t mutex;
sem_t semaphore;
int length;
int on;
```

pool_init

初始化函数进行了各项变量的初始化，使用`pthread_create`启动了所有线程池中的线程。

```
void pool_init(void)
{
    length = 0;
    on = 1;
    sem_init(&semaphore, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_create(&thread_pool[i], NULL, worker, NULL);
}
```

worker

worker为线程池中各个线程运行的代码，不断检测信号量semaphore任务队列中有无待执行的任务，如果有则调用`dequeue`和`execute`进行执行。

```
void *worker(void *param)
{
    while(1)
    {
        sem_wait(&semaphore);
        if(on == 0)
            pthread_exit(0);
        cu_task = dequeue();
        execute(cu_task.function, cu_task.data);
    }
}
```

execute

进行任务的执行

```
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}
```

pool_submit

`client.c`中我们调用了`pool_submit`完成任务提交

这里将传来的参数`enqueue`，如果返回`1`表示队列已满，那么就继续`enqueue`直到成功为止。成功后可以`sem_post(&semaphore)`使信号量`semaphore`递增，表示队列中新增加了一个任务。

```
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;
    int f = enqueue(worktodo);
    while (f)
        f = enqueue(worktodo);
    sem_post(&semaphore);
    return 0;
}
```

enqueue

完成任务队列添加任务的操作，添加的任务排在队列末尾处。如果队列已满那么返回`1`，如果未满那么在临界区进行入队操作，返回`0`。

```
int enqueue(task t)
{
    if (length == QUEUE_SIZE)
        return 1;
    pthread_mutex_lock(&mutex);
    work_queue[length] = t;
    length++;
    pthread_mutex_unlock(&mutex);
    return 0;
}
```

dequeue

完成任务队列删除任务的操作，删除的任务为队列首任务。如果队列已空那么返回空的`worktodo`，如果未满那么在临界区进行出队操作，返回出队任务`worktodo`。

```
task dequeue()
{
    task worktodo;
    if (length == 0)
    {
        worktodo.data = NULL;
```

```
        worktodo.function = NULL;
    }
    else
    {
        pthread_mutex_lock(&mutex);
        worktodo = work_queue[0];
        length--;
        for (int i = 0; i < length; ++i)
            work_queue[i] = work_queue[i + 1];
        work_queue[length].function = NULL;
        work_queue[length].data = NULL;
        pthread_mutex_unlock(&mutex);
    }
    return worktodo;
}
```

pool_shutdown

完成各个变量的delete，注意此时调用`sem_post`是为了解除在`worker`中`sem_wait`语句阻塞的线程池中的线程，使其能够执行`pthread_exit(0)`退出线程。

```
void pool_shutdown(void)
{
    on = 0;
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        sem_post(&semaphore);
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_join(thread_pool[i], NULL);
    pthread_mutex_destroy(&mutex);
    sem_destroy(&semaphore);
}
```

完整代码

```
/**
 * Implementation of thread pool.
 */

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#include "threadpool.h"

#define QUEUE_SIZE 10
#define NUMBER_OF_THREADS 3

#define TRUE 1

// this represents work that has to be
// completed by a thread in the pool
typedef struct
{
    void (*function)(void *p);
    void *data;
}
```

```
task;

task cu_task;
pthread_t thread_pool[NUMBER_OF_THREADS];
task work_queue[QUEUE_SIZE];
pthread_mutex_t mutex;
sem_t semaphore;
int length;
int on;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    if (length == QUEUE_SIZE)
        return 1;
    pthread_mutex_lock(&mutex);
    work_queue[length] = t;
    length++;
    pthread_mutex_unlock(&mutex);
    return 0;
}

// remove a task from the queue
task dequeue()
{
    task worktodo;
    if (length == 0)
    {
        worktodo.data = NULL;
        worktodo.function = NULL;
    }
    else
    {
        pthread_mutex_lock(&mutex);
        worktodo = work_queue[0];
        length--;
        for (int i = 0; i < length; ++i)
            work_queue[i] = work_queue[i + 1];
        work_queue[length].function = NULL;
        work_queue[length].data = NULL;
        pthread_mutex_unlock(&mutex);
    }
    return worktodo;
}

// the worker thread in the thread pool
void *worker(void *param)
{
    while(1)
    {
        sem_wait(&semaphore);
        if(on == 0)
            pthread_exit(0);
        cu_task = dequeue();
        execute(cu_task.function, cu_task.data);
    }
}
```

```
/**
 * Executes the task provided to the thread pool
 */
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}

/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;
    int f = enqueue(worktodo);
    while (f)
        f = enqueue(worktodo);
    sem_post(&semaphore);
    return 0;
}

// initialize the thread pool
void pool_init(void)
{
    length = 0;
    on = 1;
    sem_init(&semaphore, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_create(&thread_pool[i], NULL, worker, NULL);
}

// shutdown the thread pool
void pool_shutdown(void)
{
    on = 0;
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        sem_post(&semaphore);
    for (int i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_join(thread_pool[i], NULL);
    pthread_mutex_destroy(&mutex);
    sem_destroy(&semaphore);
}
```

结果

```
pan@pan-virtual-machine:~/桌面/osproj/5/POSIX$ ./example
I add two values 0 and 0 result = 0
I add two values 3 and 6 result = 9
I add two values 4 and 8 result = 12
I add two values 5 and 10 result = 15
I add two values 6 and 12 result = 18
I add two values 7 and 14 result = 21
I add two values 8 and 16 result = 24
I add two values 9 and 18 result = 27
I add two values 1 and 2 result = 3
I add two values 11 and 22 result = 33
I add two values 12 and 24 result = 36
I add two values 13 and 26 result = 39
I add two values 14 and 28 result = 42
I add two values 15 and 30 result = 45
I add two values 16 and 32 result = 48
I add two values 17 and 34 result = 51
I add two values 18 and 36 result = 54
I add two values 19 and 38 result = 57
I add two values 2 and 4 result = 6
I add two values 10 and 20 result = 30
pan@pan-virtual-machine:~/桌面/osproj/5/POSIX$
```

Producer-Consumer Problem

buffer implementation

在`buffer.c`中完成，定义了信号量`empty`和`full`以及互斥锁`mutex`。insert和remove操作类似Designing a Thread Pool中的写法

```c
#include "buffer.h"
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

sem_t empty;
sem_t full;
pthread_mutex_t mutex;
int length;

void buffer_init() {
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    length = 0;
}

int insert_item(buffer_item item) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    buffer[length] = item;
    length++;
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    return 0;
}

int remove_item(buffer_item *item) {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
```

```
    *item = buffer[0];
    length--;
    for (int i = 0; i < length; i++)
        buffer[i] = buffer[i + 1];
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    return 0;
}
```

main function

按照osc10e的outline进行补充即可，定义了生成者线程`producer_thread`和消费者线程`consumer_thread`

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "buffer.h"
#include <assert.h>
#include <unistd.h>

void *producer(void *param);
void *consumer(void *param);

typedef struct thread {
    pthread_t tid;
    struct thread *next;
} LinkThread;
int sleep_time, producer_num, consumer_num;

int main(int argc, char *argv[]) {

    /* 1. Get command line arguments argv[1],argv[2],argv[3] */
    assert(argc == 4);
    sleep_time = atoi(argv[1]);
    producer_num = atoi(argv[2]);
    consumer_num = atoi(argv[3]);

    /* 2. Initialize buffer */
    buffer_init();
    srand((int)time(0));
    pthread_t producer_thread[producer_num];
    pthread_t consumer_thread[consumer_num];

    /* 3. Create producer thread(s) */
    for (int i = 0; i < producer_num; i++)
        pthread_create(&producer_thread[i], NULL, producer, NULL);

    /* 4. Create consumer thread(s) */
    for (int i = 0; i < consumer_num; i++)
        pthread_create(&consumer_thread[i], NULL, consumer, NULL);

    /* 5. Sleep */
    sleep(sleep_time);

    /* 6. Exit */
    for (int i = 0; i < producer_num; i++)
```

```
        pthread_cancel(producer_thread[i]);

    for (int i = 0; i < consumer_num; i++)
        pthread_cancel(consumer_thread[i]);
    return 0;
}

void *producer(void *param) {
    buffer_item item;

    while (1) {
        /* sleep for a random period of time */
        sleep(rand() % sleep_time);
        /* generate a random number */
        item = rand();
        if (insert_item(item))
            printf("report error condition\n");
        else
            printf("producer produced %d\n", item);
    }
}

void *consumer(void *param) {
    buffer_item item;

    while (1) {
        /* sleep for a random period of time */
        sleep(rand() % sleep_time);
        if (remove_item(&item))
            printf("report error condition\n");
        else
            printf("consumer consumed %d\n", item);
    }
}
```

结果

```
pan@pan-virtual-machine:~/桌面/osproj/5/The Producer - Consumer Problem$ ./main 3 2 2
producer produced 1305590016
consumer consumed 1305590016
producer produced 315804132
consumer consumed 315804132
producer produced 1155483988
producer produced 766734814
consumer consumed 1155483988
consumer consumed 766734814
producer produced 533071958
consumer consumed 533071958
pan@pan-virtual-machine:~/桌面/osproj/5/The Producer - Consumer Problem$
```

Difficulties

- Designing a Thread Pool部分最初虽然提交了20个任务，但最后只完成了10个任务，原因是提交后没有`client.c`中进行等待就shutdown了，导致部分提交的在任务队列中的任务还没来得及执行就结束程序了。加入`sleep(1)`语句后运行正常。
- Producer-Consumer Problem部分比较简单，osc10e给出了大部分的代码实现。

Reference

- Operating System Concept 10^{th} edition
- Source code for the 10th edition of Operating System Concepts https://github.com/greggagne/osc10e